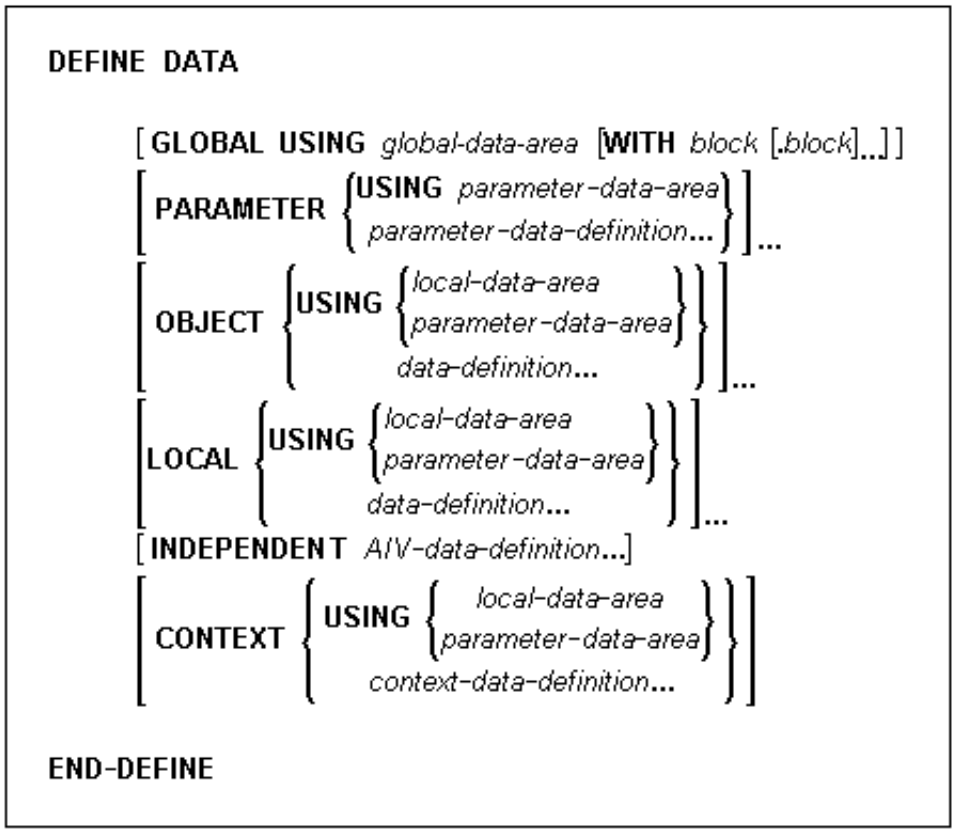
DEFINE DATA

General Syntax



Note:
If more than one clause is used, the GLOBAL, PARAMETER, OBJECT, LOCAL, INDEPENDENT and CONTEXT clauses must be specified in the order shown above.
An "empty" DEFINE DATA statement is not allowed; in other words, at least one clause (GLOBAL, PARAMETER, OBJECT, LOCAL, INDEPENDENT or CONTEXT) must be specified and at least one field must be defined.

Function

The DEFINE DATA statement is used to define the data areas which are to be used within a Natural program. When a DEFINE DATA statement is used, it must be the first statement of the program/routine.

DEFINE DATA in Structured Mode

In structured mode, all variables to be used must be defined in the DEFINE DATA statement; they must not be defined elsewhere in the program.

DEFINE DATA in Reporting Mode

In reporting mode, the DEFINE DATA statement is not mandatory since variables may be defined in the body of the program. However, if a DEFINE DATA LOCAL statement is used in reporting mode, variables (except AIVs) must not be defined elsewhere in the program; and if a DEFINE DATA INDEPENDENT statement is used in reporting mode, application-independent variables (AIVs) must not be defined elsewhere in the program.

DEFINE DATA OBJECT

DEFINE DATA OBJECT is used in conjunction with NaturalX. It is described in the NaturalX documentation.

data areas

Natural supports three types of data areas:

- global data area
- parameter data area
- local data area

global data area

A global data area contains data elements which can be referenced by more than one programming object (as described in section Object Types of the Natural Programming Guide). The global data area and the objects which reference it must be contained in the same library (or in a steplib). No more than one global data area is allowed per DEFINE DATA statement.

parameter data area

A parameter data area contains data elements which are used as parameters in a subprogram, external subroutine or dialog. Parameter data elements must not be assigned initial or constant values, and they must not have EM, HD or PM definitions. Parameter data elements can also be defined within the subprogram/subroutine itself. Parameters can also be defined within a helproutine.

local data area

A local data area contains data elements which are to be used in a single Natural module. (Local data can also be defined directly within a program or routine.) A data area defined using DEFINE DATA LOCAL may be a parameter data area.

All three types of data areas can be created and maintained by using the data area editor.

block

Data blocks can overlay one another during program execution, thereby saving storage space.

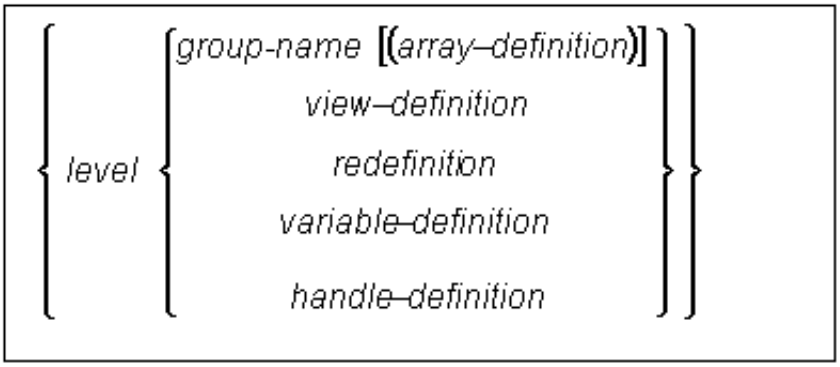
The maximum number of block levels is 8 (including the master block).

.block

.block notations(s) specify the block(s) which are used in the program.

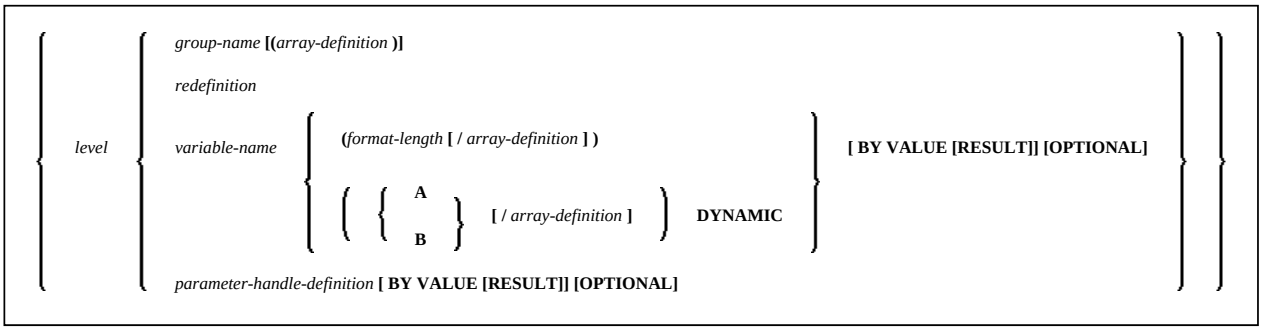
For further information on data blocks, see the section Data Blocks in the Natural Programming Guide.

data-definition



level	Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading "0" is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only 1 field) by using the group name. This provides a convenient and efficient method of referencing a series of consecutive fields. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped. A view-definition must always be defined at level 1.
group-name	The name of a group. The name must adhere to the rules for defining a Natural variable name.

parameter-data-definition



level	This is the same as under data-definition.
group-name	This is the same as under data-definition.
variable-name	This is the same as under variable-definition.
format-length	This is the same as under variable-definition.
DYNAMIC	A parameter may be defined as DYNAMIC. For more information on processing dynamic variables, see Large and Dynamic Variables/Fields. Depending on whether call-by-reference or call-by-value is used, the appropriate transfer mechanism is applicable. For further information, see the CALLNAT statement.
BY VALUE	Without BY VALUE (default), a parameter is passed to a subprogram/subroutine by reference (that is, via its address); therefore a field specified as parameter in a CALLNAT/PERFORM statement must have the same format/length as the corresponding field in the invoked subprogram/subroutine. With BY VALUE, a parameter is passed to a subprogram/subroutine by value; that is, the actual parameter value (instead of its address) is passed. Consequently, the field in the subprogram/subroutine need not have the same format/length as the CALLNAT/PERFORM parameter. The formats/lengths must only be data transfer compatible. For data transfer compatibility, the Rules for Arithmetic Assignment/Data Transfer apply (see Statement Usage Related Topics). BY VALUE allows you, for example, to increase the length of a field in a subprogram/subroutine (if this should become necessary due to an enhancement of the subprogram/subroutine) without having to adjust any of the objects that invoke the subprogram/subroutine. For parameter definitions for dialogs (under Windows), the following applies: - Without BY VALUE, a parameter, as specified in the inline definition of a dialog's parameter data area, is transferred via its address (by reference); the format and length of the parameter in an OPEN DIALOG or SEND EVENT statement, for example, must match the format and length of the parameter in the inline parameter data definition of the dialog. You can use a parameter by reference in the before open and after open event handlers and in all other events if the used parameters are transferred in the SEND EVENT statement triggering this event. - With BY VALUE, a parameter is transferred via its value; format and length do not have to match; the parameter in the OPEN DIALOG or SEND EVENT statement must be data transfer compatible with the parameter of the dialog.
BY VALUE RESULT	While BY VALUE applies to a parameter passed to a subprogram/subroutine, BY VALUE RESULT causes the parameter to be passed by value in both directions; that is, the actual parameter value is passed from the invoking object to the subprogram/subroutine and, on return to the invoking object, the actual parameter value is passed from the subprogram/subroutine back to the invoking object. With BY VALUE RESULT, the formats/lengths of the fields concerned must be data transfer compatible in both directions. Note: BY VALUE RESULT cannot be used in dialogs.
OPTIONAL	For a parameter defined without OPTIONAL (default), a value *must* be passed from the invoking object. For a parameter defined with OPTIONAL, a value can--but need not be--passed from the invoking object to this parameter. In the invoking object, the notation *n*X is used to indicate parameters which are skipped, that is, for which no values are passed. With the SPECIFIED Option you can find out at run time whether an optional parameter has been defined or not.

Example of BY VALUE:

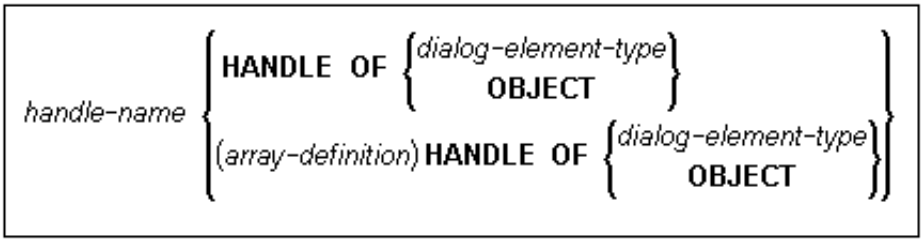
```
* Program
DEFINE DATA LOCAL
  1 #FIELDA (P5)
  ...
END-DEFINE
...
CALLNAT 'SUBR01' #FIELDA
...
```

```
* Subroutine SUBR01
DEFINE DATA PARAMETER
  1 #FIELDB (P9) BY VALUE
END-DEFINE
...
```

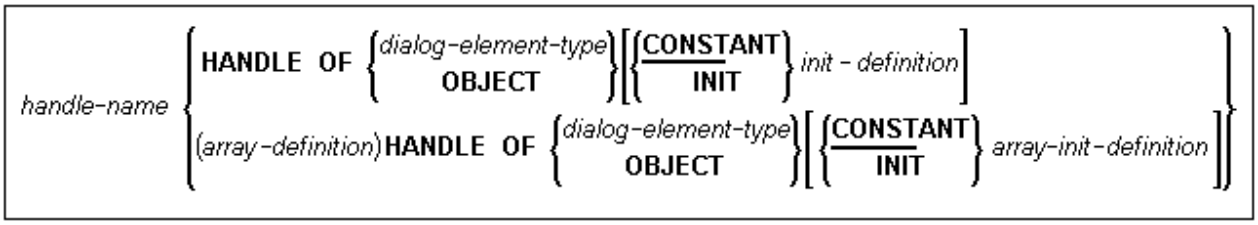
Example of BY VALUE for Dialog:

```
/*Example of three parameters not passed BY VALUE:
1 #A (A10)   /* Parameter Data Definition
1 #B (A20)   /* of the called Dialog
1 #C (A30)   /*
OPEN DIALOG 'MYDIALOG' #DLG$WINDOW WITH #X #Y #Z /* #X has to be A10,#Y has to
                                                 /* be A20,and #Z has to be A30
/*Example of three parameters passed BY VALUE:
1 #A (A10) BY VALUE   /* Parameter Data Definition
1 #B (A20) BY VALUE   /* of the called Dialog
1 #C (A30) BY VALUE   /*
OPEN DIALOG 'MYDIALOG' #DLG$WINDOW WITH #X #Y #Z  /* #X may be A1, #Y may be
                                                  /* A100,and #Z may be A253
```

parameter-handle-definition



handle-definition



The use of "*handle-definition*" with "*dialog-element-type*" is only possible under Windows.

handle-name	The name to be assigned to the handle; the naming conventions for user-defined variables apply (see the section Naming Conventions under User-Defined Variables).
dialog-element-type	The type of dialog element (only possible under Windows). Its possible values are the values of the TYPE attribute. For details, see the sections **Dialog Elements** and **Attributes** of the Natural Dialog Component Reference documentation for Windows.
OBJECT	Is used in conjunction with NaturalX as described in the NaturalX documentation.

The HANDLE definition in the DEFINE DATA statement is generated automatically on the creation of a dialog element or dialog.

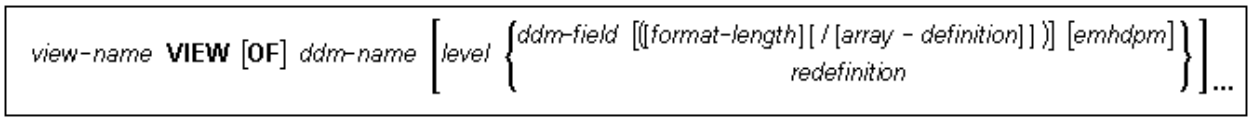
After having defined a handle, you can use the handle-name in any statement to query, set or modify attribute values for the defined dialog-element-type (see the section Event-Driven Programming Techniques in the Natural Programming Guide).

Examples of handle-definitions:

```
1 #SAVEAS-MENUITEM HANDLE OF MENUITEM
1 #OK-BUTTON (1:10) HANDLE OF PUSHBUTTON
```

Note:
If you use "block" structures, a HANDLE OF OBJECT may only be defined in the master block, but not in any subordinate blocks.

view-definition



A *view-definition* is used to define a view as derived from a DDM.

In a parameter data area, *view-definition* is not permitted.

view-name	The name to be assigned to the view. Rules for Natural variable names apply.
ddm-name	The name of the DDM from which the view is to be taken.
level	Level number is a 1- or 2-digit number in the range from 01 to 99 (the leading "0" is optional) used in conjunction with field grouping. Fields assigned a level number of 02 or greater are considered to be a part of the immediately preceding group which has been assigned a lower level number. The definition of a group enables reference to a series of fields (may also be only one field) by using the group name. This provides a convenient and efficient method of referencing a series of consecutive fields. A group may consist of other groups. When assigning the level numbers for a group, no level numbers may be skipped.
ddm-field	The name of a field to be taken from the DDM. When you define a view for a HISTOGRAM statement, the view must contain only the descriptor for which the HISTOGRAM is to be executed.
format-length	Format and length of the field. If omitted these are taken from the DDM. In structured mode, the definition of format and length must be the **same** as those in the DDM. In reporting mode, the definition of format and length must be **compatible** with those in the DDM.
array-definition	Depending on the mode used, arrays (periodic-group fields, multiple-value fields) may have to contain information about their occurrences. For more information, see the explanation below the table.
DYNAMIC	Defines a view field as DYNAMIC. For more information on processing dynamic variables, see the section Large and Dynamic Variables/Fields.

array-definition

In structured mode, if a field is used in the view which represents an array, you **must** include a corresponding index range in the view definition.

Structured Mode:

Adabas: If array fields defined in a DDM are to be used inside a view, these fields must contain an explicit array definition. The array dimensions must match the corresponding DDM definition exactly (considering the inheritance of array dimensions of previous groups). Only the array bounds may differ. The number of occurrencies must not pass the maximum of 191 and the index range must be within the defined index range of the DDM (periodic groups and multiple fields: (1:191)).

SQL: No array definitions allowed.

XML: If array fields defined in a DDM are to be defined inside a view, these fields must contain an explicit array definition. The array dimensions must match the corresponding DDM definition exactly. Only the index range may differ. The number of occurrences must not exceed the defined one and the index range must be within the defined index range. If X-arrays are defined in DDM, they also may be used inside the view.

Note:
A DDM of type XML is only valid under Windows and UNIX.

The following table shows which view definition is allowed according to the DDM definition:

Note:
In the table, Z: is an integer variable, and X1, X2, Y1, Y2, Y: are constants or constant expressions.

	view-definition	
DDM definition	**allowed**	**not allowed**
A(*:X2)	A(*:Y2) Y2≤X2 A(Y1:Y2) Y2>Y1 Y2≤X2 A(Z:Z+Y) Y≥0	A(*:*) A(Y1:*)
A(X1:*)	A(Y1:*) Y1≥X1 A(Y1:Y2) Y2≥X1, Y1≥X1 A(Z:Z+Y) Y≥0	A(*:*) A(*:Y2)
A(X1:X2)	A(Y1:Y2) Y2<Y1 A(Z:Z+Y) 0≤Y≥X2-X1+1	A(*:*) A(Y1:*) A(*:Y2)

Examples:

```
DEFINE DATA LOCAL
1 EMP1 VIEW OF EMPLOYEES
2 NAME(A20)
2 ADDRESS-LINE(A20 / 1:2)
/* or
1 EMP2 VIEW OF EMPLOYEES
2 NAME
2 ADDRESS-LINE(1:2)
/* or
1 EMP3 VIEW OF EMPLOYEES
2 NAME
2 ADDRESS-LINE(2)
/* or
1 #K (I4)
1 EMP4 VIEW OF EMPLOYEES
2 NAME
2 ADDRESS-LINE(#K:#K+1)
END-DEFINE
END
```

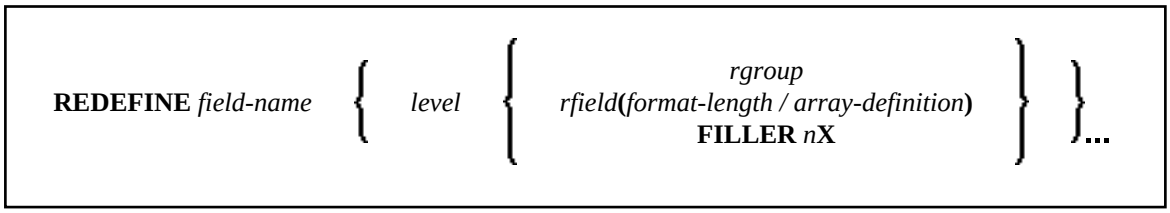
Reporting Mode

In this mode, the same rules are valid as for structured mode. However, there is one exception: the specification of array bounds is not a must. The index range may be omitted completely. In this case the index range for the missed dimensions is set to (1:1).

Examples:


```
DEFINE DATA LOCAL
1 EMP1 VIEW OF EMPLOYEES
2 NAME(A30)
2 ADDRESS-LINE(A35 / 5:10)
/* or
1 EMP2 VIEW OF EMPLOYEES
2 NAME
2 ADDRESS-LINE(A40)
/* or
1 EMP3 VIEW OF EMPLOYEES
2 NAME
2 ADDRESS-LINE
/* or
1 #K (I4)
1 EMP4 VIEW OF EMPLOYEES
2 NAME
2 ADDRESS-LINE(#K:#K+1)
END-DEFINE
END
```

redefinition



A redefinition may be used to redefine a group, a view, a DDM field or a single field/variable (that is a scalar or an array).

field-name	The name of the group, view, DDM field or single field that is being redefined.
level	This is the same as under data-definition.
rgroup	The name of the group resulting from the redefinition. **Note:** In a "redefinition" within a "view-definition", the name of "rgroup" must be different from any field name in the underlying DDM.
rfield	The name of the field resulting from the redefinition. **Note:** In a "redefinition" within a "view-definition", the name of "rfield" must be different from any field name in the underlying DDM.
format-length	The format and length of the rfield.
FILLER*n*X	With this notation, you define n filler bytes - that is, segments which are not to be used - in the field that is being redefined. The definition of trailing filler bytes is optional.

Restrictions Regarding Handles, X-Arrays and Dynamic Variables

Handles, X-arrays and dynamic variables cannot be redefined and cannot be contained in a redefinition clause.
A group that contains a Handle, X-array or a dynamic variable can only be redefined up to - but not including or beyond - the element in question.

Note:
In a "parameter-data-definition", a "redefinition" of groups is only permitted within a REDEFINE block; otherwise internal errors might occur when passing parameters between the calling program and the called subprogam.

REDEFINE - Example 1:

```
DEFINE DATA LOCAL
  01 #VAR1 (A15)
  01 #VAR2
     02 #VAR2A (N4.1) INIT <0>
     02 #VAR2B (P6.2) INIT <0>
  01 REDEFINE #VAR2
     02 #VAR2RD (A10)
END-DEFINE
...
```

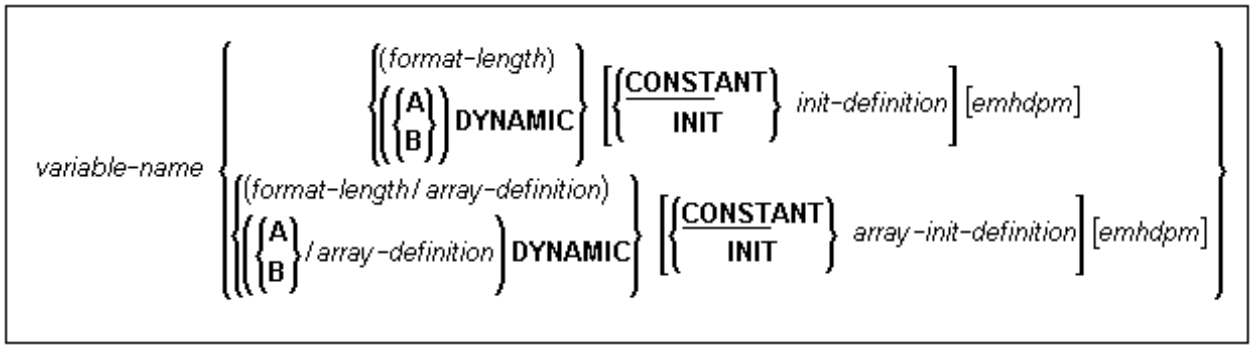
REDEFINE - Example 2:

```
DEFINE DATA LOCAL
01 MYVIEW VIEW OF STAFF
   02 NAME
   02 BIRTH
   02 REDEFINE BIRTH
      03 BIRTH-YEAR  (N4)
      03 BIRTH-MONTH (N2)
      03 BIRTH-DAY   (N2)
END-DEFINE
...
```


REDEFINE - Example 3:

```
DEFINE DATA LOCAL
1 #FIELD (A12)
1 REDEFINE #FIELD
  2 #RFIELD1 (A2)
  2 FILLER 2X
  2 #RFIELD2 (A2)
  2 FILLER 4X
  2 #RFIELD3 (A2)
END-DEFINE
...
```

variable-definition



With a variable definition a single field/variable (that is a scalar or an array) is defined.

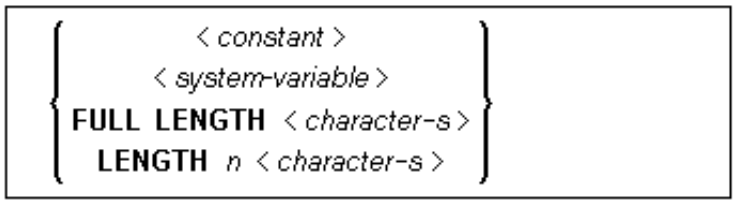
variable-name	The name to be assigned to the variable. Rules for Natural variable names apply. For information on naming conventions for user-defined variables, see the section Statement Usage Related Topics.
format-length	The format and length of the field. For information on format/length definition of user-defined variables, see the section Statement Usage Related Topics.
DYNAMIC	A field may be defined as DYNAMIC. For more information on processing dynamic variables, see the section Large and Dynamic Variables/Fields.
CONSTANT	The variable/array is to be treated as a named constant. The constant value(s) assigned will be used each time the variable/array is referenced. The value(s) assigned cannot be modified during program execution. **Note:** For reasons of internal handling, it is not allowed to mix variable definitions and constant definitions within one group definition; that is, a group may contain either variables only or constants only.
INIT	The variable/array is to be assigned an initial value. This value will also be used when this variable/array is referenced in a RESET INITIAL statement.

If no INIT or CONSTANT specification is supplied, a field will be initialized with a default initial value depending on its format (see Default Initial Values below).

Default Initial Values

Format	Default Initial Value
B, F, I, N, P	0
A	blank
L	F(ALSE)
D	D’ ’
T	T’00:00:00’
C	(AD=D)
GUI Handle	NULL-HANDLE
Object Handle	NULL-HANDLE

init-definition



Note:
The INIT and CONST clauses cannot be used with X-arrays.

constant	The constant value with which the variable is to be initialized; or the constant value to be assigned to the field. See the section Statement Usage Related Topics for further information on constants.
system-variable	The initial value for a variable may also be the value of a Natural system variable. **Note:** When the variable is referenced in a RESET INITIAL statement, the system variable is evaluated again; that is, it will be reset not to the value it contained when program execution started but to the value it contains when the RESET INITIAL statement is executed.
FULL LENGTH **LENGTH** *n*	As initial value, a variable can be filled, entirely or partially, with a specific single character or string of characters (only possible for alphanumeric variables). With the "FULL LENGTH" option, the entire field will be filled with the specified *character* or *characters*. With the "LENGTH *n*" option, the first *n* positions of the field will be filled with the specified *character* or *characters*. *n* must be a numeric constant.

Example of System Variable as Initial Value:

```
DEFINE DATA LOCAL
1 #MYDATE (D) INIT <*DATX>
END-DEFINE
```

Example of FULL LENGTH:

In this example, the entire field will be filled with asterisks.

```
DEFINE DATA LOCAL
1 #FIELD (A25) INIT FULL LENGTH <'*'>
END-DEFINE
```

Example of LENGTH *n*:

In this example, the first 4 positions of the field will be filled with exclamation marks.

```
DEFINE DATA LOCAL
1 #FIELD (A25) INIT LENGTH 4 <'!'>
END-DEFINE
```

array-definition

{ *bound* **[** **:***bound* **]** **},...** *3*

You define the lower and upper bound of a dimension in an array-definition. You can define up to 3 dimensions for an array.

If only one bound is specified, the specified bound is assumed to be the upper bound and the lower bound is assumed to be 1.

bound

A bound can be one of the following:

- a numeric integer constant;
- a previously defined named constant;
- (for database arrays) a previously defined user-defined variable; or
- * defines an extensible bound, otherwise known as an X-array.
  Note:
  X-arrays are only available under Windows and UNIX.

If at least one bound in at least one dimension of an array is specified as extensible, that array is then called an X-array (eXtensible array). Only one bound (either upper or lower) may be extensible in any one dimension, but not both. Multi-dimensional arrays may have a mixture of constant and extensible bounds, e.g. #a(1:100, 1:*).

Example:

```
DEFINE DATA LOCAL
1 #ARRAY1(I4/1:10)
1 #ARRAY2(I4/10)
1 #X-ARRAY3(I4/1:*)
1 #X-ARRAY4(I4/*,1:5)
1 #X-ARRAY5(I4/*:10)
1 #X-ARRAY6(I4/1:10,100:*,*:1000)
END-DEFINE
```

If the following table you can see the bounds of the arrays in the above program more clearly.

	Dimension1 Lower bound	Dimension1 Upper bound	Dimension2 Lower bound	Dimension2 Upper bound	Dimension3 Lower bound	Dimension3 Upper bound
#ARRAY1	1	10	-	-	-	-
#ARRAY2	1	10	-	-	-	-
#X-ARRAY3	1	eXtensible	-	-	-	-
#X-ARRAY4	1	eXtensible	1	5	-	-
#X-ARRAY5	eXtensible	10	-	-	-	-
#X-ARRAY6	1	10	100	eXtensible	eXtensible	1000

Examples of Array Definitions:

```
   #ARRAY2(I4/10)                  /* a one-dimensional array
   #X-ARRAY4(I4/*,1:5)             /* a two-dimensional array
   #X-ARRAY6(I4/1:10,100:*,*:1000)  /* a three-dimensional array
```


Variable Arrays in a Parameter Data Area

In a parameter data area, you may specify an array with a variable number of occurrences. This is done with the index notation "`1:V`". For example:

```
#ARRAYX (A5/1:V)
```

```
#ARRAYY (I2/1:V,1:V)
```

An array that is defined with index "1:V" must not be redefined or be the result of a redefinition. As the number of occurrences is not known at compilation time, it must not be referenced with the index notation (*) in any statement, except ADD, COMPRESS, COMPUTE, DISPLAY, DIVIDE, EXAMINE, IF, MULTIPLY, RESET, SUBTRACT.

A variable array can only be referenced either in its entirety (that is, all its occurrences) or as a scalar value (that is, a single occurrence). For example:

```
#ARRAYX (*)
#ARRAYY (*,*)
#ARRAYX (1)
#ARRAYY (5,#FIELDX)
```

A partial range of a variable array must not be referenced:

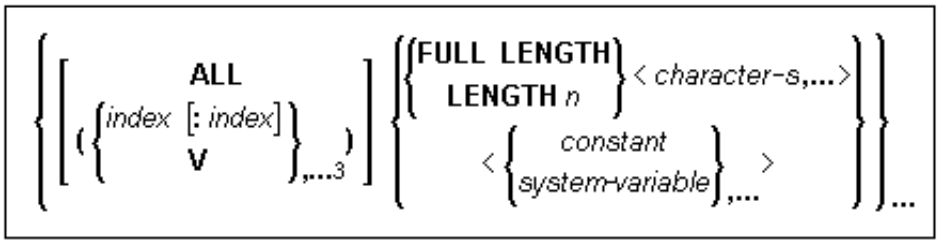
```
#ARRAYY (1,*)  /* not allowed
```

To avoid runtime errors, the maximum number of occurrences of such an array should be passed to the subprogram/subroutine via another parameter.

Notes:
If a parameter data area that contains an index "1:V" is used as a local data area (that is, specified in a DEFINE DATA LOCAL statement), a variable named "V" must have been defined as CONSTANT.
In a dialog, an index "1:V" cannot be used in conjunction with BY VALUE.

See also the system variable *OCCURRENCE in the Natural System Variables documentation.

array-init-definition



With this clause, you define the initial/constant values for an array.

For a redefined field, an *array-init-definition* is not permitted.

ALL	All occurrences in all dimensions of the array are initialized with the same value.
index	Only the array occurrences specified by the *index* are initialized. If you specify *index*, you can only specify one value with *constant*; that is, all specified occurrences are initialized with the same value.
V	This notation is only relevant for multidimensional arrays if the occurrences of one dimension are to be initialized with *different* values. "V" indicates an index range that comprises all occurrences of the dimension specified with "V"; that is, all occurrences in that dimension are initialized. Only *one* dimension per array may be specified with "V". The occurrences are initialized occurrence by occurrence with the values specified for that dimension. The number of values must not exceed the number of occurrences of the dimension specified with "V".
constant	The constant (value) with which the array is to be initialized (INIT), or the constant to be assigned to the array (CONSTANT). See the section Statement Usage Related Topics for further information on defining constants. **Note:** Occurrences for which no values are specified, are initialized with a default value.
system-variable	The initial value for an array may also be the value of a Natural system variable. **Note:** Multiple constant values/system variables must be separated either by the input delimiter character (as specified with the session parameter ID) or by a comma. A comma must not be used for this purpose, however, if the comma is defined as decimal character (with the session parameter DC).
FULL LENGTH **LENGTH***n*	As initial value, it is also possible to have an array filled, entirely or partially, with a specific single character or string of characters (only possible for alphanumeric arrays). With "FULL LENGTH", the entire array occurrence(s) are filled with the specified *character* or *characters*. With "LENGTH *n*", the first *n* positions of the array occurrence(s) are filled with the specified *character* or *characters*. A *system-variable* must not be specified with "FULL LENGTH" or "LENGTH *n*". Within one *array-init-definition*, only either "FULL LENGTH" or "LENGTH *n*" may be specified; both notations must not be mixed.

Example of LENGTH n for Array:

In this example, the first 5 positions of each occurrence of the array will be filled with "NONON".

```
DEFINE DATA LOCAL
1 #FIELD (A25/1:3) INIT ALL LENGTH 5 <'NO'>
...
END-DEFINE
```

Numerous examples of assigning initial values to arrays are provided in the Natural Programming Guide.

emhdpm

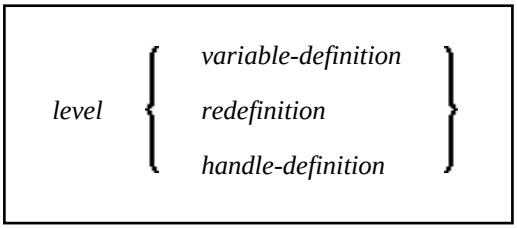
([EM = *value* **] [HD** = *' value '* **] [PM** = *value* **])**

With this option, additional parameters to be in effect for the field/variable may be defined.

EM=value	This parameter may be used to define an edit mask. See the session parameter EM in the Natural Parameter Reference documentation.
HD='value'	This parameter may be used to define the header to be used as the default header for the field (see the DISPLAY statement).
PM=value	This parameter may be used to set the print mode, which indicates how fields are to be output. See the session parameter PM in the Natural Parameter Reference documentation.

If for a database field you specify neither an edit mask (EM=) nor a header (HD=), the default edit mask and default header as defined in the DDM will be used.
However, if you specify one of the two, the other's default from the DDM will `not` be used.

AIV-data-definition



Additional Rules

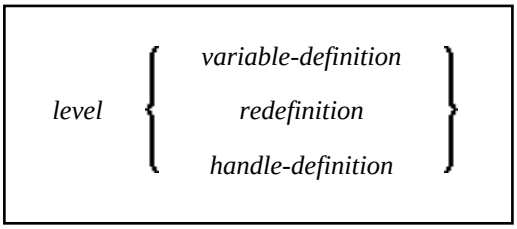
- An application-independent variable must be defined at level 01. Other levels are only used in a redefinition.
- The CONSTANT clause must not be used in this context.
- The first character of the name must be a "+". Rules for Natural variable names apply. For information on naming conventions for user-defined variables, see the section Naming Conventions under User-Defined Variables
- The fields resulting from the redefinition must not be application-independent variables, that is their name must not start with a ’+’. These fields are treated as local variables.

DEFINE DATA INDEPENDENT is used to define application-independent variables (AIVs).

An application-independent variable is referenced by its name, and its content is shared by all programming objects executed within one application that refer to that name. The variable is allocated by the first executed programming object that references this variable and is deallocated by the LOGON command or a RELEASE VARIABLES statement. The optional INIT clause is evaluated in each executed programming object that contains this clause (not only in the programming object that allocates the variable).

level
variable-definition
redefinition
handle-definition

context-data-definition



Additional rules:

- A context variable must be defined at level 01. Other levels are only used in a redefinition.
- The CONSTANT clause must not be used in this context.
- The fields resulting from the redefinition are not considered a context variable. These fields are treated as local variables.

DEFINE DATA CONTEXT is used in conjunction with Natural RPC. It is used to define variables known as context variables, which are meant to be available to multiple remote subprograms within one conversation, without having to explicitly pass the variables as parameters with the corresponding CALLNAT statements.

A context variable is referenced by its name, and its content is shared by all programming objects executed in one conversation that refer to that name. The variable is allocated by the first executed programming object that contains the definition of the variable and is deallocated when the conversation ends. The optional INIT clause is evaluated in each executed programming object that contains this clause (not only in the programming object that allocates the variable). This is different to the way the INIT works for global variables.

Context variables can also be used in a non-conversational CALLNAT. In this case, the context variables only exist during a single invocation of this CALLNAT but the variables can be shared with all its callees.

A context variable is not shared with subprograms that are called within the conversation. If such a subprogram or one of its callees references a context variable, a separate storage area is allocated for this variable.

For further information, see Defining a Conversation Context in the Natural RPC documentation.

level
variable-definition
redefinition
handle-definition

Qualifying Data Structures

To identify a field when referencing it, you may qualify the field; that is, before the field name, you specify the name of the level-1 data element in which the field is located and a period.

If a field cannot be identified uniquely by its name (for example, if the same field name is used in multiple groups/views), you must qualify the field when you reference it.

The combination of level-1 data element and field name must be unique (see first example below).

The qualifier must be a level-1 data element (see second example below).

Example:

```
DEFINE DATA LOCAL
1 FULL-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
1 OUTPUT-NAME
  2 LAST-NAME (A20)
  2 FIRST-NAME (A15)
END-DEFINE
...
MOVE FULL-NAME.LAST-NAME TO OUTPUT-NAME.LAST-NAME
...
```

Example:

```
DEFINE DATA LOCAL
1 GROUP1
  2 SUB-GROUP
    3 FIELD1 (A15)
    3 FIELD2 (A15)
END-DEFINE
...
MOVE 'ABC' TO GROUP1.FIELD1
...
```

Note:
If you use the same name for a user-defined variable and a database field (which you should not do anyway), you must qualify the database field when you want to reference it; because if you do not, the user-defined variable will be referenced instead.

Example 1

```
/* EXAMPLE 'DDAEX1': DEFINE DATA
/************************************************************
DEFINE DATA LOCAL
01 #VAR1 (A15)
01 #VAR2
   02 #VAR2A (N4.1) INIT <1111>
   02 #VAR2B (N6.2) INIT <22222>
01 REDEFINE #VAR2
   02 #VAR2C (A2)
   02 #VAR2D (A2)
   02 #VAR2E (A6)
END-DEFINE
/************************************************************
WRITE NOTITLE '=' #VAR2A / '=' #VAR2B /
              '=' #VAR2C / '=' #VAR2D / '=' #VAR2E
/************************************************************
END
```

```
#VAR2A:  1111.0
  #VAR2B:  222222.00
  #VAR2C: 11
  #VAR2D: 11
  #VAR2E: 022222
```

Example 2

```
/* EXAMPLE 'DDAEX2': DEFINE DATA (ARRAY DEFINITION/INITIALIZATION)
/************************************************************
DEFINE DATA LOCAL
01 #VAR1 (A1/1:2,1:2) INIT (1,V) <'A','B'>
01 #VAR2 (N5/1:2,1:3) INIT (1,2) <200>
01 #VAR3 (A1/1:4,1:3) INIT (V,2:3) <'W','X','Y','Z'>
END-DEFINE
/************************************************************
WRITE NOTITLE '=' #VAR1 (1,1) '=' #VAR1 (1,2)
            / '=' #VAR1 (2,1) '=' #VAR1 (2,2)
/************************************************************
WRITE ///     '=' #VAR2 (1,1) '=' #VAR2 (1,2)
            / '=' #VAR2 (2,1) '=' #VAR2 (2,2)
/************************************************************
WRITE ///     '=' #VAR3 (1,1) '=' #VAR3 (1,2) '=' #VAR3 (1,3)
WRITE       / '=' #VAR3 (2,1) '=' #VAR3 (2,2) '=' #VAR3 (2,3)
WRITE       / '=' #VAR3 (3,1) '=' #VAR3 (3,2) '=' #VAR3 (3,3)
WRITE       / '=' #VAR3 (4,1) '=' #VAR3 (4,2) '=' #VAR3 (4,3)
/************************************************************
END
```



```
#VAR1: A #VAR1: B
  #VAR1:   #VAR1:



  #VAR2:      0 #VAR2:    200
  #VAR2:      0 #VAR2:      0



  #VAR3:   #VAR3: W #VAR3: W

  #VAR3:   #VAR3: X #VAR3: X

  #VAR3:   #VAR3: Y #VAR3: Y

  #VAR3:   #VAR3: Z #VAR3: Z
```

Example 3

```
/* EXAMPLE 'DDAEX3': DEFINE DATA (VIEW DEFINITION, REDEFINE ARRAY)
/************************************************************************
DEFINE DATA LOCAL
1 EMPLOY-VIEW VIEW OF EMPLOYEES
  2 NAME
  2 ADDRESS-LINE (A20/2)
  2 PHONE
1 #ARRAY (A75/1:4)
1 REDEFINE #ARRAY
  2 #ALINE (A25/1:4,1:3)
1 #X (N2) INIT <1>
1 #Y (N2) INIT <1>
END-DEFINE
/************************************************************************
FORMAT PS=20
LIMIT 5
FIND EMPLOY-VIEW WITH NAME = 'SMITH'
 MOVE NAME            TO #ALINE (#X,#Y)
 MOVE ADDRESS-LINE(1) TO #ALINE (#X+1,#Y)
 MOVE ADDRESS-LINE(2) TO #ALINE (#X+2,#Y)
 MOVE PHONE           TO #ALINE (#X+3,#Y)
 IF  #Y = 3
  RESET INITIAL #Y
  PERFORM PRINT
 ELSE
  ADD 1 TO #Y
 END-IF
 AT END OF DATA
  PERFORM PRINT
 END-ENDDATA
END-FIND
/************************************************************************
DEFINE SUBROUTINE PRINT
WRITE NOTITLE (AD=OI) #ARRAY(*)
RESET #ARRAY(*)
SKIP 1
END-SUBROUTINE
/************************************************************************
END
```

```
SMITH                    SMITH                      SMITH
  ENGLANDSVEJ 222          3152 SHETLAND ROAD         14100 ESWORTHY RD.
                           MILWAUKEE                  MONTERREY
  554349                   (414)877-4563              (408)994-2260

  SMITH                    SMITH
  5 HAWTHORN               2307 DARIUS LANE
  OAK BROOK                TAMPA
  (312)150-9351            (813)131-4010
```

Example 4

```
   /* EXAMPLE 'DDAEX4': DEFINE DATA (GLOBAL, PARAMETER AND LOCAL AREAS)
   /************************************************************************
   /* MAIN PROGRAM
   /************************************************************************
   DEFINE DATA GLOBAL USING GLOBAL-1
               LOCAL
               1 #FIELD1 (A10)
               1 #FIELD2 (N5)
   END-DEFINE
   /************************************************************************
   /* ...
   CALLNAT 'SUBP1' #FIELD1 #FIELD2
   /* ...
   END
```

```
   /* SUBPROGRAM 'SUBP1'
   DEFINE DATA PARAMETER
   1 #FIELDA (A10)
   1 #FIELDB (N5)
   END-DEFINE
   /************************************************************************
   /* ...
   END
```

Example 5

```
   * EXAMPLE 'DDAEX5': DEFINE DATA (INITIALIZATION)
   ********************************************
   DEFINE DATA LOCAL
    1 #START-DATE (D)   INIT <*DATX>
    1 #UNDERLINE  (A50) INIT FULL LENGTH <'_'>
    1 #SCALE      (A65) INIT LENGTH 65 <'....+..../'>
   END-DEFINE
   *
   WRITE NOTITLE #START-DATE (DF=L)
       / #UNDERLINE
       / #SCALE
   END
```



```
1999-01-19
  _____________________________________________
  ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+
```

Example 6

```
   /* EXAMPLE 'DDAEX6': DEFINE DATA (VARIABLE ARRAY)
   /*****************************************************************
   DEFINE DATA
      PARAMETER
        01 #STRING (A1/1:V)
        01 #MAX (P3)
      LOCAL
         01 #I (P3)
   END-DEFINE
   /*****************************************************************
   F0R #I = 1 TO #MAX
     IF #STRING (#I) < H'40'
       MOVE H'40' TO #STRING (#I)
     END-IF
   END-FOR
   END
```

Example 7

```
DEFINE DATA LOCAL
  1 #MyHomePage (A4096)        /* alpha variable with max. 4096 characters
  1 #MyStream (B1000000/1:10)  /* binary array with 10 occurrences and max. 1000000 bytes per occ.
  1 #MyDynHomePage (A) DYNAMIC /* dynamic alpha variable
  1 #MyDynStream (B) DYNAMIC   /* dynamic binary variable
  END-DEFINE
```

